Integration, the VLSI Journal 74 (2020) 19–31

Contents lists available at ScienceDirect

Integration, the VLSI Journal

journal homepage: www.elsevier.com/locate/vlsi

# Logarithm-approximate floating-point multiplier is applicable to power-efficient neural network training☆

TaiYu Cheng [a,*], Yukata Masuda [b], Jun Chen [a], Jaehoon Yu [c], Masanori Hashimoto [a]

[a] Department of Information Systems Engineering, Osaka University, Japan
[b] Center for Embedded Computing Systems, Graduate School of Informatics, Nagoya University, Japan
[c] Institute of Innovative Research, Tokyo Institute of Technology, Japan

ARTICLE INFO

*Keywords:*
Approximate computing
Neural network
Training engine
Floating-point unit
Logarithm multiplier
GPU design

ABSTRACT

Recently, emerging "edge computing" moves data and services from the cloud to nearby edge servers to achieve short latency and wide bandwidth, and solve privacy concerns. However, edge servers, often embedded with GPU processors, highly demand a solution for power-efficient neural network (NN) training due to the limitation of power and size. Besides, according to the nature of the broad dynamic range of gradient values computed in NN training, floating-point representation is more suitable. This paper proposes to adopt a logarithm-approximate multiplier (LAM) for multiply-accumulate (MAC) computation in neural network (NN) training engines, where LAM approximates a floating-point multiplication as a fixed-point addition, resulting in smaller delay, fewer gates, and lower power consumption. We demonstrate the efficiency of LAM in two platforms, which are dedicated NN training hardware, and open-source GPU design. Compared to the NN training applying the exact multiplier, our implementation of the NN training engine for a 2-D classification dataset achieves 10% speed-up and 2.3X efficiency improvement in power and area, respectively. LAM is also highly compatible with conventional bit-width scaling (BWS). When BWS is applied with LAM in five test datasets, the implemented training engines achieve more than 4.9X power efficiency improvement, with at most 1% accuracy degradation, where 2.2X improvement originates from LAM. Also, the advantage of LAM can be exploited in processors. A GPU design embedded with LAM executing an NN-training workload, which is implemented in an FPGA, presents 1.32X power efficiency improvement, and the improvement reaches 1.54X with LAM + BWS. Finally, LAM-based training in deeper NN is evaluated. Up to 4-hidden layer NN, LAM-based training achieves highly comparable accuracy as that of the accurate multiplier, even with aggressive BWS.

## 1. Introduction

For enhancing our daily life with artificial intelligence (AI), machine learning (ML) is currently adopted and executed everywhere, even on end devices such as smartphones, Internet-of-Things (IoT) sensors, and cameras. The generated data from devices need to be collected and aggregated as the samples for learning processing and analysis. Conventionally, the data is transferred to the cloud since the learning and analysis processes require high computational capability and large memory capacity. However, the data moving to the cloud involves challenges in terms of latency, bandwidth, and privacy concerns. Some applications like computer vision and natural language processing highly demand local and real-time services, and then edge computing emerges as a solution. In edge computing, the edge servers, typically embedded with GPU processors, are placed near the end devices, and they process and analyze the data independently from the cloud or before sending the data to the cloud [1,2]. Compared with ML training in the cloud, training in the edge servers may provide tailored ML models without security risk [1]. On the other hand, due to the size and power limitations, power-efficient training is demanded and explored [3,4].

Neural network (NN) is one of the most widely-used techniques in machine learning [5]. A feedforward NN model is composed of a few to hundreds of layers, each of which includes a number of neurons. The neurons are connected layer by layer through synaptic weights. The

☆ This work is supported by Grant-in-Aid for Scientific Research (B) from JSPS under Grant 19H04079.
* Corresponding author.
*E-mail addresses:* t-cheng@ist.osaka-u.ac.jp (T. Cheng), masuda@ertl.jp (Y. Masuda), j-chen@ist.osaka-u.ac.jp (J. Chen), yu.jaehoon@artic.iir.titech.ac.jp (J. Yu), hasimoto@ist.osaka-u.ac.jp (M. Hashimoto).

https://doi.org/10.1016/j.vlsi.2020.05.002
Received 18 November 2019; Received in revised form 2 May 2020; Accepted 5 May 2020
Available online 14 May 2020

synaptic weights are optimized to provide sufficiently high accuracy through the computationally expensive training phase. Hardware NN system is mainly categorized into two types. The first one is the inference engine that processes a network with given pre-trained weights, and the latter is the training engine that has the additional capability of weight optimization in the training phase. Regardless of the inference or training engine, multiply-accumulate (MAC) arithmetic computation is the primary operation. The rapidly increasing trend of NN size to deal with more intricate and sophisticated problems explodes the amount of MAC computation, resulting in a strong demand for dedicated hardware engines.

Inference engines in several studies exploit inherent error-tolerant property in machine learning and introduce approximate computing (AC) techniques for gaining performance and reducing cost [6–9]. Among various AC techniques proposed for inference engines, bit-width scaling (BWS), which reduces the bit width of data representation, is the most popular and powerful way that trades computation reduction with accuracy degradation [6,10–12]. Even binarized neural networks are studied [13].

In contrast to inference, training engines need to perform more arithmetic computation with a wider dynamic range since the gradient, which is numerically computed and used to guide the weight update, spreads in a broad dynamic range [10]. A simple example can help us understand this property. Fig. 1 plots the distribution density of the gradient values found when training a NN for MNIST dataset [24]. The gradient values spread from $2^{-47}$ to $2^{6}$. If adopting fixed-point expression, more than 50 bits are required to cover this range, while floating-point expression spends only a few bits for exponents and some extra bits for fraction parts to cover this wide range. Thus, adopting floating-point units (FPUs) is beneficial for training engines to accommodate such gradient computation.

Nevertheless, FPUs are known as power-hungry and area-expensive units [14]. Specifically, in the neural network algorithm that mainly consists of MAC computations, floating-point multiplication is the most power-hungry and space-demanding arithmetic operators. Fig. 2 exemplifies the power and area benchmarking between a 32-bit floating-point multiplier and an adder synthesized for the same clock frequency. The figure shows that the floating-point multiplier consumes 3.01X power and 1.75X area. The MAC operation requires the same number of multiplication and addition, indicating that the power for MAC operation is mostly consumed by the multiplier. Consequently, massive MAC computations in NN training deteriorate the area and power efficiency. Therefore, power-efficient floating-point multiplication is highly demanded in training engine development.

In this paper, we perform that the logarithm-approximate multiplier (LAM), which approximates floating-point multiplication to fixed-point addition, benefits to NN training and improves the power efficiency of

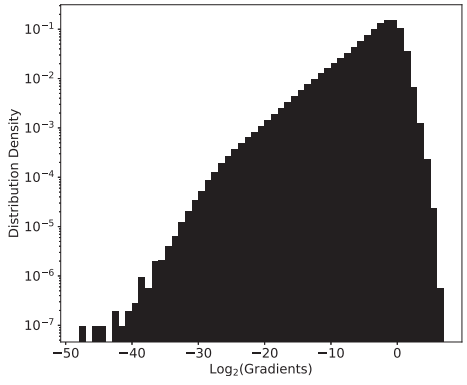



**Fig. 1.** Distribution density of gradients observed when a NN is trained for MNIST dataset [24]. *x*-axis represents the gradients in log scale with base 2, and *y*-axis is the normalized distribution density.

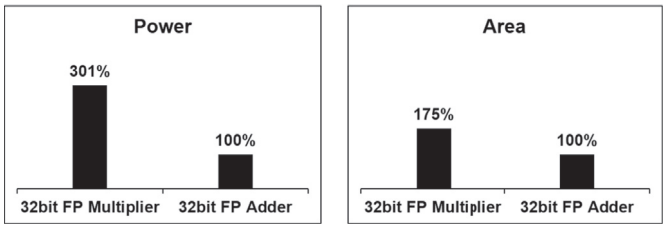



**Fig. 2.** Hardware benchmarking for 32-bit floating-point multiplier and adder synthesized for the same clock frequency. Both power and area values are normalized by those of the adder, respectively.

massive MAC computation involved in NN training under floating-point format. We also show that LAM is useful even when the BWS is already implemented in training, and hence power efficiency is further enhanced. These advantages are quantitatively evaluated through the experiments with dedicated training hardware. Experimental results show 2.5X power reduction by LAM and 4.9X reduction by LAM + BWS while sustaining the accuracy, where 2.2X reduction originates from LAM. These results are reported in our preliminary work [15].

In this work, we newly evaluate whether solo or hybrid usages of exact and approximate multipliers in training and testing phases affect the classification accuracy. Experimental results reveal that adopting approximate multipliers (LAMs) in both training and testing phases induces no significant accuracy degradation, and then there is no need to rely on accurate multipliers. Next, we conduct additional experiments for evaluating the applicability of LAM and LAM + BWS to open-source GPU design, on which NN training programs are executed. The power reductions thanks to LAM and LAM + BWS are measured with an FPGA implementation of the GPU design, and they are 1.32X and 1.54X compared to the original design. Finally, we extend the NN depth, up to 4-hidden layers, and show that solo-LAM training achieves highly comparable results with solo-EFM training. This trend sustains even when BWS is aggressively adopted as long as the acceptable training accuracy is obtained.

The rest of the paper is organized as follows. Section 2 reviews the NN with its training and related works. Section 3 introduces LAM and discusses its approximation error. Experimental results of adopting LAM in dedicated training engines are presented in Section 4. Section 5 provides the environmental setup and measurement results of LAM-based NN training with FPGA implementation of an open-source GPU design. Section 6 applies LAM-based training to deeper NNs and provides evaluation results. Finally, Section 7 concludes this paper.

## 2. Preliminaries, related work and research motivation

### 2.1. Basics of neural network

Fig. 3a illustrates a multilayer perceptron (MLP) structure, which is known as a basic feedforward NN [18]. For the sake of clarity, the structure contains only 1 hidden layer, while the number of hidden layers can be extended to form deeper NNs. The state of each neuron in the network is computed from all the states of neurons in the previous layer and then propagates its state to the next layer. Taking the example in Fig. 3a, since all the states are pre-determined in the input layer $I$, the state computation starts at each neuron in the hidden layer (neurons are denoted as $H_1$, $H_2$, …, $H_h$). Each neuron in the hidden layer computes the sum of all the states of neurons in the input layer ($I_1$, $I_2$, …, $I_i$) multiplied with corresponding synaptic weights ($W^H$), passes the sum with a bias term ($B^H$) through a non-linear activation function to determine its state, and then propagates the state to the output layer. This operation is repeated at the neurons in the output layer ($O_1$, $O_2$, …, $O_o$) again, but here the sum of all the states of neurons in the hidden layer is computed with weights ($W^O$) and bias ($B^O$) to determine the states. The procedure finishes once all the states of neurons in the output layer are determined.

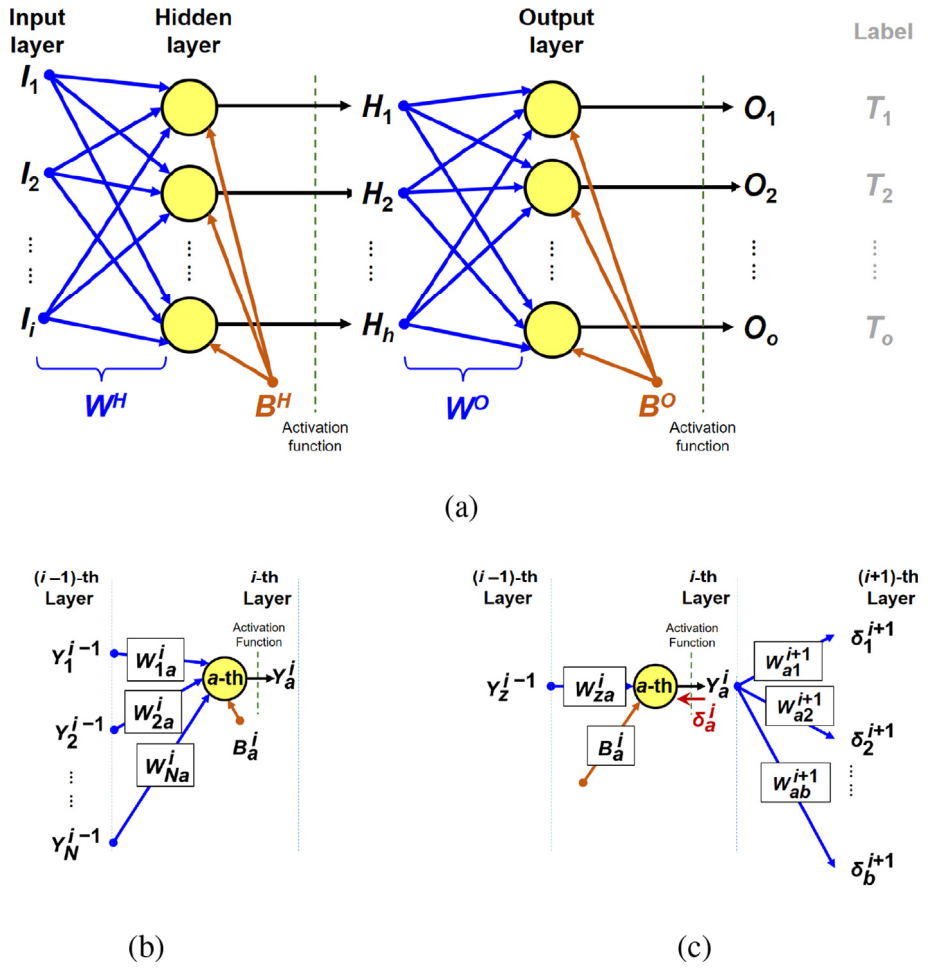



**Fig. 3.** Feed-forward neural network. (a) is a schematic of a feed-forward neural network with 1 hidden layer. (b) and (c) are the schematics for illustrating forward- and back-propagation at the $a$-th neuron in $i$-th layer.

A basic unit for expressing the above operation is shown in Fig. 3b. Suppose there is the $a$-th neuron in the $i$-th layer, its state $Y_a^i$ can be computed by the following formula:

$$Y_a^i = \mathrm{Act}\left(\sum_{k=1}^{N} W_{ka}^i Y_k^{i-1} + B_a^i\right), \tag{1}$$

where $Y_k^{i-1}$ denotes the state of the $k$-th neuron in the $(i-1)$-th layer. $W_{ka}^{i-1}$ represents the synaptic weight connecting from the $k$-th neuron in the $(i-1)$-th layer, and $B_a^i$ denotes the bias term for the $a$-th neuron in the $i$-th layer. Act(.) means the activation function, which usually allows passing $\geq 0$ values or limits the values between $-1$ and 1. This procedure, so-called forward propagation, keeps going until the states of all the neurons in the output layer are determined, which is the core and dominant operation that an inference engine with pre-trained weights executes.

On the other hand, training NN aims at finding a set of synaptic weights and bias values to minimize the loss function (Loss), which is usually defined as the error squared between the state from the forward propagation results in output layer $O$ and target label $T$ ($T_1$, $T_2$, …, $T_o$) [18]. For illustration purposes, we describe the standard back-propagation. At the beginning of the training phase, all the weights are randomly initialized, the biases may be initially set to 0, and then forward propagation is launched. The next step is to reduce the Loss function according to the contribution of each synaptic weight ($W$) and bias ($B$), which can be obtained through computing their gradient, i.e. $\partial\mathrm{Loss}/\partial W$ and $\partial\mathrm{Loss}/\partial B$. Based on the computed gradients, each synaptic weight and bias can be numerically updated during each iteration. Let us take Fig. 3c as an example. Suppose a synaptic weight $W_{za}^i$ connects the $z$-th neuron in the $(i-1)$-th layer (state $= Y_z^{i-1}$) with the $a$-th neuron in the $i$-th layer (state $= Y_a^i$) and a bias $B_a^i$ is for the $a$-th neuron in the $i$-th layer. Then, $W_{za}^i$ and $B_a^i$ are updated by:

$$W_{za}^i += -\eta \frac{\partial \mathrm{Loss}}{\partial W_{za}^i} \text{ where } \frac{\partial \mathrm{Loss}}{\partial W_{za}^i} = \delta_a^i Y_z^{i-1}, \tag{2}$$

$$B_a^i += -\eta \frac{\partial \mathrm{Loss}}{\partial B_a^i} \text{ where } \frac{\partial \mathrm{Loss}}{\partial B_a^i} = \delta_a^i. \tag{3}$$

The gradient terms $\partial\mathrm{Loss}/\partial W_{za}^i$ and $\partial\mathrm{Loss}/\partial B_a^i$ in (2) and (3) share the same term $\delta_a^i$ while $\partial\mathrm{Loss}/\partial W_{za}^i$ further includes the term $Y_z^{i-1}$. Basically, the gradient terms would decay during the weight and bias update, and thus these are also called gradient decent method. $\eta$ is the learning rate, and $\delta_a^i$ is conditionally formulated as follows. If the $i$-th layer is the output layer, then $\delta_a^i$ is:

$$\delta_a^i = \mathrm{Act}'(O_a)(O_a - T_a), \tag{4}$$

where $O_a$ represents the state computed through forward propagation, $T_a$ means its target state, and Act' means the derivative of activation function. If the $i$-th layer is not the output layer, then referring to Fig. 3c, $\delta_a^i$ is expressed as:

$$\delta_a^i = \mathrm{Act}'(Y_a^i)\Big(\sum_{n=1}^{b} W_{an}^{i+1}\delta_n^{i+1}\Big), \tag{5}$$

where $W_{an}^{i+1}$ denotes the synaptic weight to the $n$-th neuron in the $(i+1)$-th layer. $\delta_n^{i+1}$ can be recursively computed through (4) and (5). Note that, with (4) and (5), the output loss is propagating backward from the output layer, and thus this procedure is named as back propagation [18]. In addition, (5) indicates that the $\delta_a^i$ in the non-output layer needs to compute all the weighted sum of $\delta_n^{i+1}$, meaning that MAC computation is also primary in back propagation. Therefore, the training phase executes huge amount of MAC computations during the iteration loops of forward and back propagation. In addition, the gradient terms have a large dynamic range, and thus adopt floating-point arithmetic is beneficial in training.

### 2.2. Related work and research motivation

Several papers propose to introduce bit-width scaling [7,8] and approximate multipliers [19,20] into NN. The former uses fewer bits in computation while the latter aims at reducing the resource for the multiplication operation.

Logarithm based multiplier is a typical type of approximate multiplier since logarithm converts multiplication to addition. State-of-the-art logarithm multipliers, such as [33,34] adopt this property to approximate fixed-point multiplication in cooperation with a dedicated and efficient error-fixing solution to mitigate the calculation error compared to exact multiplication. Some researchers exploit the logarithm-based multiplier in NN. Reference [20] applies iterative logarithmic multipliers (ILM), which is a preliminary version of [33,34], to error-tolerant training algorithm while [19,31,32] convert the multiplicand and multiplier into log domain to do multiplication as an addition. References [31,32] perform even addition and activation function in log domain to execute the complete training in log domain. The above studies are all based on fixed-point representations of the original value and its log value. Recent works [21,23] propose to adopt logarithm-based floating-point multipliers in NN, but [23] only adopts it in an inference engine. As for floating-point training engines [6,10,11,35], study BWS only. Although [21] uses the logarithm-based multiplier in training, their focus is to provide a run-time configurable solution that can switch exact and logarithm-based multipliers. Once an error that is larger than a pre-determined criterion is detected in the logarithm-based multipliers [21], automatically switches back to the exact multiplier. The same first author of [21] has an earlier publication that introduced two-stage training, in which the approximate training is allowed in the early stage while in the late stage, the accurate training is demanded [22]. Consequently [21,22], still rely on the exact multiplier in training and solo logarithm-based training is beyond their interest. Besides, the compatibility with BWS and the efficacy in deeper NNs are not addressed.

In summary, previous studies for floating-point NN training intensively focus on BWS, and it has left the space for evaluating the efficacy of the logarithm-based multiplier in training. Also, BWS is still the primary choice in the training engine design, and hence the compatibility between the logarithm-based multiplier and BWS must be investigated. Taking into account the tremendous number of MAC operations in training engines and the limited power budget for edge computing, exploring a useful approximate technique for pursuing higher power efficiency and examining its compatibility with BWS could give a useful implication for training-engine designers, which is the objective of this work. Besides, in addition to realizing a training engine through application specific integrated circuit (ASIC), GPU-level applicability for LAM is also one of our interests.

## 3. Logarithm approximate multiplier

Logarithm-approximate multiplier, LAM in short, is developed by Ref. [16]. With an approximation, a floating-point value in linear domain can be regarded as its value taken by the logarithm of base 2 in fixed-point format. Thanks to the log-domain property, floating-point multiplication can be approximated to fixed-point addition. This section introduces LAM and analyzes its approximation error.

### 3.1. Floating-point multiplication

The floating-point format consists of three parts to represent a value in scientific notation: one bit for sign, several bits for exponent, and the remaining bits for fraction. When a value $i$ is represented in the floating-point format containing $N$ bits for exponent and $M$ bits for fraction, then $i$ is expressed as:

$$i = (-1)^{S_i}\cdot 2^{(E_i-\mathrm{bias})}\cdot(1+F_i/2^M). \tag{6}$$

$S_i$ is 0 or 1, where 0/1 means $i$ is a positive/negative value. $F_i$ is the fraction part when $i$ is converted to base-2 scientific notation with multiplying $2^M$, and hence $0 \le F_i/2^M \le 1$. $E_i$ is the exponent value that includes a bias $= 2^{(N-1)} - 1$.

Fig. 4 explains the multiplication of two floating-point values $C = A \times B$, where the sign parts ($S_A$, $S_B$ and $S_C$), exponential parts ($E_A$, $E_B$ and $E_C$) and fraction parts ($F_A$, $F_B$ and $F_C$) in the floating-point representation are processed individually.

$$S_\oplus = S_A \oplus S_B, \tag{7}$$

$$E_+ = E_A + E_B, \tag{8}$$

$$F_\times/2^M = (1+F_A/2^M)\times(1+F_B/2^M), \tag{9}$$

where $(1 + F_A/2^M)$ is obtained by appending 1 to the binary representation of $F_A$, and supposing the binary point exists between 1 and $F_A$. Then, the multiplication result is expressed by:

$$S_C = S_\oplus, \tag{10}$$

$$E_C = \begin{cases} E_+ - \mathrm{bias} & F_\times/2^M < 2, \\ E_+ - \mathrm{bias} + 1 & \text{otherwise}, \end{cases} \tag{11}$$

$$F_C = \begin{cases} F_\times - 2^M & F_\times/2^M < 2, \\ F_\times/2 - 2^M & \text{otherwise}. \end{cases} \tag{12}$$

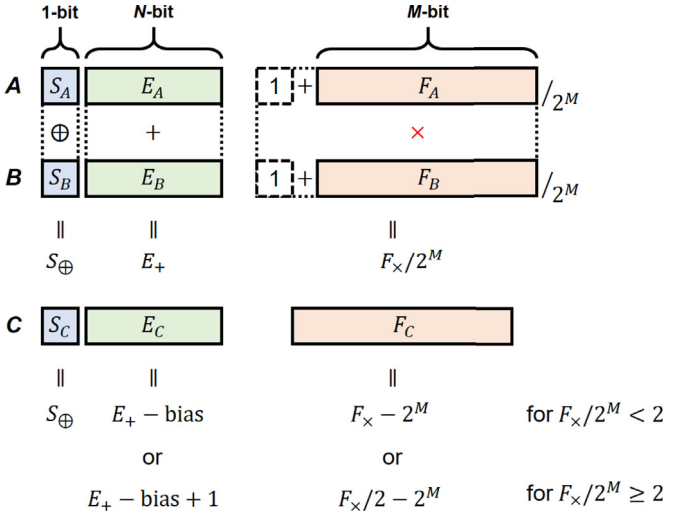



**Fig. 4.** Operation of exact floating-point multiplier. ($S_\otimes$, $E_+$, $F_\times$) are individually calculated by XOR, addition, and multiplication based on the sign, exponent, and fraction part of multiplicand and multiplier, respectively. Then, $E_C$ and $F_C$ are the conditional formula according to $F_\times$.

In the hardware point of view, the multiplier for $F_\times$ consumes considerable power and area, and it often limits the speed since it contains many full adders or similar logics in both width and depth.

### 3.2. Logarithm-approximate multiplier

Focusing on a positive floating-point number $i$, we simplify (6) for the sake of clarity in the following discussion.

$$i = 2^e(1+f), \tag{13}$$

where $e$ replaces $E_i$ −bias and $f$ replaces $F_i/2^M$. When converting $i$ into log domain, (13) becomes

$$\log_2 i = e + \log_2(1+f) \cong e + f, \tag{14}$$

where the approximated representation in the right term utilizes the approximation below.

$$\log_2(1+x) \cong x, \text{for} 0 \le x \le 1. \tag{15}$$

When approximating $\log_2(1+x)$ at $x=0$, (15) becomes $1.44x$. On the other hand, when we intend to approximate $\log_2(1+x)$ in the region of $0 \le x \le 1$, the approximation to $1.0x$ is also possible as shown in Fig. 5. With this approximation, (13) and (14) make any manipulation unnecessary to approximate a floating-point value $i$ to a fixed-point value $\log_2 i$.

The logarithmic domain is beneficial in multiplication, as mentioned in Section 2.2, since it can convert multiplication to addition. Fig. 6 illustrates the approximate multiplier that we name as logarithm-approximate multiplier (LAM). To compute $C = A \times B$, according to LAM, the following primary computations are performed.

$$S_\oplus = S_A \oplus S_B, \tag{16}$$

$$E_+ = E_A + E_B, \tag{17}$$

$$F_+ / 2^M = F_A / 2^M + F_B / 2^M, \tag{18}$$

where we can find the calculation of fraction parts has changed from multiplication to addition thanks to the property of log-domain while (16) and (17) are still identical to (7) and (8). Making a mantissa from a fraction is also excluded, and the fraction part is directly used for computation. Then, the final multiplication result is expressed by:

$$S_C = S_\oplus, \tag{19}$$

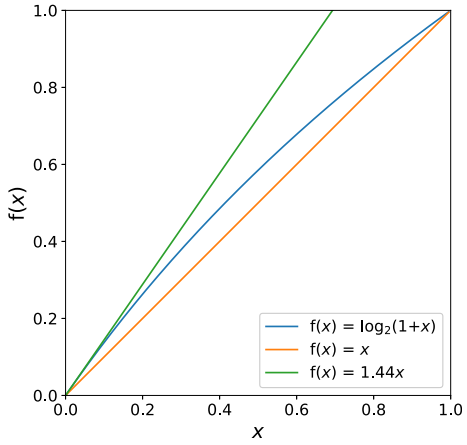



**Fig. 5.** Curves of functions $\log_2(1+x)$, $1.0x$, and $1.44x$. Taking into the entire range of $0 \le x \le 1$, $1.0x$ is a possible approximation of $\log_2(1+x)$, while $1.44x$ is better at the point of $x=0$.

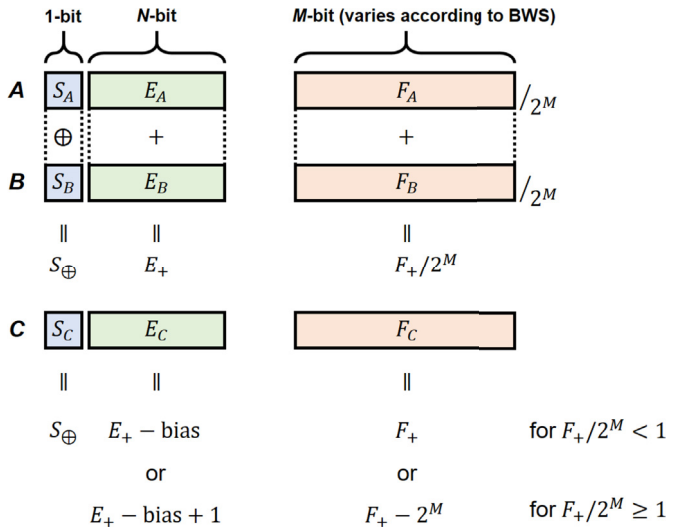



**Fig. 6.** Operation of logarithm-approximate multiplier (LAM). $S_\otimes$ and $E_+$ are identical to those of the exact floating-point multiplier, but $F_+$ is directly computed by adding the fractions without making their mantissas. $E_C$ and $F_C$ are expressed by the conditional formula regarding $F_+$.

$$E_C = \begin{cases} E_+ - \text{bias} & F_+/2^M < 1, \\ E_+ - \text{bias} + 1 & \text{otherwise}, \end{cases} \tag{20}$$

$$F_C = \begin{cases} F_+ & F_+/2^M < 1, \\ F_+ - 2^M & \text{otherwise}. \end{cases} \tag{21}$$

Note that equation (19) to calculate sign part $S_C$ is identical to (10) and totally isolated from computing $E_C$ and $F_C$. Therefore, LAM can perform multiplication irrelevantly to positive and negative values using the equations from (16) to (21). In our work, BWS is achieved by varying the variable $M$ denoted in Fig. 6, which can be easily integrated in both exact multiplication and LAM.

Although $E_C$ and $F_C$ in (20) and (21) are conditional equations, they can be efficiently computed in hardware implementations. We concatenate $\{E_A, F_A\}$ and $\{E_B, F_B\}$, respectively, add them and subtract the bias term followed by $M$-bits of 0s, as illustrated in Fig. 7. Then, the overflow coming from $F_A + F_B$ can directly add a carry to $E_A + E_B$. Finally, $E_C$ and $F_C$ are exactly the first $N$ bits and the last $M$ bits of the computed result.

The approximation error of LAM can be analyzed by directly comparing the computations of exact multiplication and LAM. Again, for the sake of clarity, let us just focus on two positive floating-point values $A$ and $B$ of $A = 2^{e_A}(1+f_A)$ and $B = 2^{e_B}(1+f_B)$, where the notations of $e_A$, $e_B$, $f_A$, and $f_B$ refer to (13). The result of exact multiplication $C = A \times B$ is:

$$C_{exact} = 2^{e_A+e_B}(1+f_A)(1+f_B). \tag{22}$$

When computing $A \times B$ by LAM, the expression is:

$$C_{LAM} = \begin{cases} 2^{e_A+e_B}(1+f_A+f_B) & f_A+f_B < 1, \\ 2^{e_A+e_B+1}(f_A+f_B) & \text{otherwise}. \end{cases} \tag{23}$$

By comparing the expressions under different conditions, relative error of approximation, *ErrLAM*, can be derived as a conditional function of $f_A$ and $f_B$:

$$ErrLAM = \frac{C_{exact} - C_{LAM}}{C_{exact}} = \frac{Err}{(1+f_A)(1+f_B)} \tag{24}$$

where

$$Err = \begin{cases} f_A f_B & f_A+f_B < 1, \\ (1-f_A)(1-f_B), & \text{otherwise}. \end{cases} \tag{25}$$

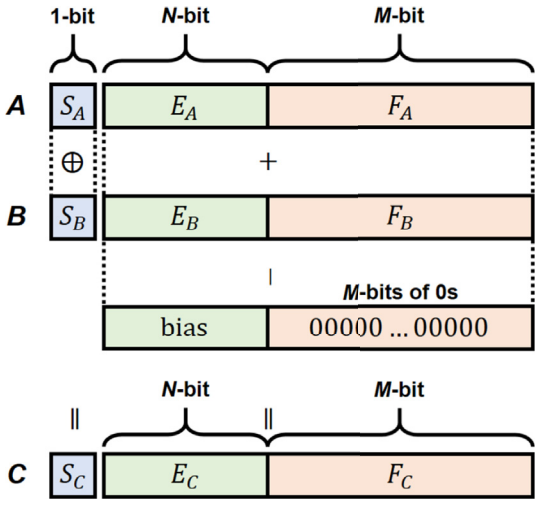



**Fig. 7.** Algorithm for implementing LAM in hardware. $E_A$ and $F_A$ are concatenated, and $E_B$ and $F_B$ as well. Then, we can directly sum up them and subtract the bias term that is followed by $M$-bits of 0s to compute $E_C$ and $F_C$.

Here, both $f_A$ and $f_B$ are between 0 and 1, and hence *ErrLAM* is always above 0. Fig. 8 shows a contour map of *ErrLAM* as a function of $f_A$ and $f_B$. The maximum value of *ErrLAM* is about 11.1% when both $f_A$ and $f_B$ equal to 0.5. Note that *ErrLAM* is not affected by the exponents of $A$ and $B$ since $e_A$ and $e_B$ are all canceled out when dividing (23) by (22). Similarly, the sign values do not change the absolute value of *ErrLAM* neither. Fig. 8 also indicates that, as long as either $f_A$ or $f_B$ is closed to 0 or 1, the approximate error is well suppressed. The advantage of LAM in terms of the speed, power, and area will be discussed in Section 4.1, and the impact of the approximation error on the NN training will be investigated in Sections 4.3 and 4.4.

## 4. Experimental results for dedicated hardware design

This section shows the advantage of LAM as an arithmetic unit and demonstrates the impact of LAM in NN training engine on the classification accuracy and hardware resource.

### 4.1. LAM performance

We first evaluate the performance of LAM as a multiplier by

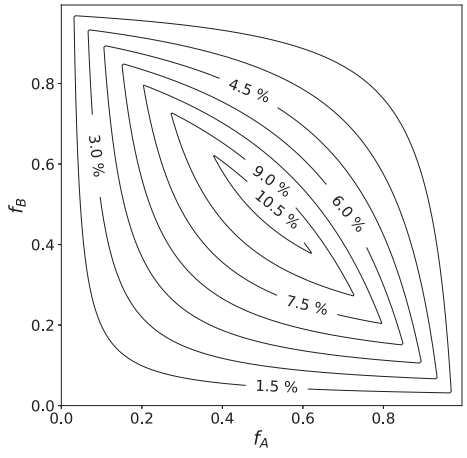



**Fig. 8.** Contour plot of *ErrLAM*, which represents the relative approximation error between LAM and exact floating-point multiplier. The error depends on the fraction values ($f_A$, $f_B$) of the multiplicand $A$ and multiplier $B$ under base-2 scientific notation, where $0 \leq f_A, f_B < 1$.

comparing two multipliers; one is LAM, and another is the baseline exact floating-point multiplier, denoted as EFM. LAM is manually implemented at RTL with Verilog while EFM directly adopts Synopsys DesignWare IP (DW_fp_mult) for functional credibility and sophisticated quality. The two designs are synthesized by Synopsys Design Compiler with an open-source 45 nm Nangate cell library. We examine their speed, power, and area during benchmarking. The power and area evaluation are performed in two scenarios, one is at their individual highest speed and the other one is at a uniform speed at which the slowest multiplier can be synthesized.

The evaluation results are shown in Fig. 9. For the sake of clarity, all the results are normalized by that of 32-bit EFM. Fig. 9 shows LAM achieves 2.5X speed compared with EFM and consumes 5.9X less power and 8.3X less area at that speed while 12.5X less power and 7.7X less area at the uniform speed in case of 32-bit floating-point expression (where sign-exponent-fraction = 1-8-23). The results of 16-bit version (1-5-10) are also presented. An interesting observation is that 32-bit LAM operates faster and consumes less power and area than even 16-bit EFM. Thus, LAM can improve energy efficiency remarkably. The following sections evaluate the impact of LAM on NN training with BWS in terms of the NN classification accuracy and hardware performance.

### 4.2. Experimental setup for NN training

Table 1 lists the datasets [17,24–27] used for the experiments in this paper and the numbers of neurons in the NNs for each dataset. For the experiments, we prepare a 3-layer structure, which means 1 hidden layer (each layer is denoted as $I$, $H$, $O$), and adopt ReLU (Rectified Linear Unit, $y = \max(x, 0)$) and sigmoid function ($y = 1/(1 + \exp(-x))$) as the activation function of hidden and output layers, respectively. Cross entropy is chosen as the loss function since it can combine with the sigmoid function to simplify (4) to a simple linear relation, $\delta_a^i = O_a - T_a$ [28]. We also adopt stochastic gradient decent with mini batch, which updates weights and bias after accumulating $\partial\text{Loss}/\partial W$ and $\partial\text{Loss}/\partial B$ from a batch size of training data, and apply learning rate decay, which gradually declines learning rate along with iterations, as well for better convergent speed during the training.

Besides, the hardware implementation of non-linear sigmoid function is costly, and hence we adopt PLAN (Piecewise Linear Approximation of a Nonlinear function) function to approximate sigmoid which is proposed in Ref. [29]. PLAN function requires only shifters and adders, and hence it is friendly to hardware implementation. Fig. 10a lists the conditional equations used for sigmoid approximation, and Fig. 10b shows the curves of the original and PLAN sigmoid functions. We can observe that PLAN is well correlated with the original sigmoid function.

Fig. 11 plots the diagram of our dedicated NN training hardware engine containing arithmetic units such as ⊗ for multipliers and ⊕ for adders. Forward- block computes (1), Back- block calculates (4) and (5), and then Updating block computes (2) and (3). Here $a$, $b$, and $c$ denote the numbers of neurons in Layer $I$, $H$, $O$, ReLU′ means the derivative function of ReLU and Reg is register. The subtractor and PLAN function are annotated as adders since their functionality is similar while ReLU is drawn by ○. In Updating block, we use the accumulators to add up the gradients $\delta$ for the number of batch size and then update the associated weights and biases based on the accumulated gradients with multiplying a learning rate $\eta$.

### 4.3. Evaluation for FOURCLASS dataset

Now, we evaluate the NN training engine with FOURCLASS dataset [17], which is a simple 2-D classification problem and its training results can be graphically illustrated with the boundary line separating the two classified groups. Fig. 12 shows the training results of 32-bit and 16-bit floating-point cases, where the samples in the same group share the same color. Note that the training and testing phase are both carried out

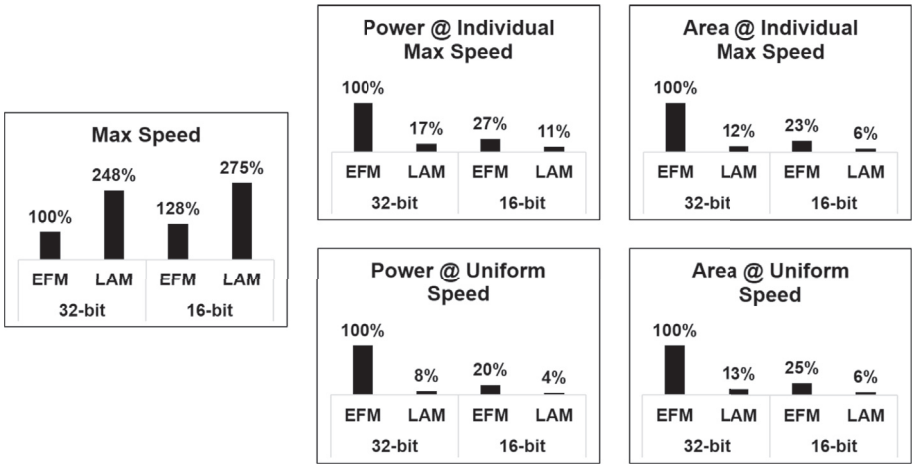



**Fig. 9.** Speed, power, and area benchmarking for synthesized LAM and EFM. There is one speed comparison and two scenarios for power and area under the max speed circuit or the uniform speed circuit. All the values are normalized to EFM 32-bit case.

**Table 1**
NN structures for testcases based on 1-hidden layer.

| Dataset | #Neurons (*I*, *H*, *O*) | batch size |
|---|---|---|
| FOURCLASS | (2,8,2) | 100 |
| MNIST | (400,300,10) | 100 |
| HARS | (561,40,6) | 40 |
| ISOLET | (617,100,26) | 60 |
| CNAE-9 | (856,100,9) | 40 |

with solo EFM or solo LAM. Although the boundary lines of EFM and LAM show some discrepancy, both of them successfully distinguish the groups of samples. In 32-bit case, both EFM and LAM achieve 100% classification accuracy. In 16-bit case, which corresponds to BWS adoption, LAM experiences 0.4% accuracy drop while it is negligibly small. This result suggests LAM is compatible with BWS.

Fig. 13 shows the hardware evaluation results, to which a normalization to 32-bit EFM engine is applied in the same way as Section 4.1. The training engine adopting LAM gains 10% speed-up over the EFM engine, where this speed-up is smaller than Fig. 9 since the floating-point adder is now a speed-limiting module. In addition to the speed improvement, the maximum speed circuit achieves 2.1X (= 100%/47%) power and 2.2X (= 100%/45%) area efficiency enhancements. On the other hand, in the uniform speed circuit, 2.3X (= 100%/43%) power and 2.3X (= 100%/44%) area efficiency enhancements are attained by LAM. These results confirm that the multiplier is so power and space demanding that reducing its computational cost improves the power and area efficiency considerably. Furthermore, considering 16-bit LAM as a scenario that LAM and BWS are applied together, 32% speed-up, 5.9X power, and 5.3X area efficiency enhancements are attained in the maximum speed circuit, where 5% (=132% − 127%) speed-up, 2.5X (=100%/17% − 100%/30%) power reduction, and also 2.1X (=100%/19% − 100%/32%) area savings originate from LAM. The benefits of training with LAM are quantitatively clarified.

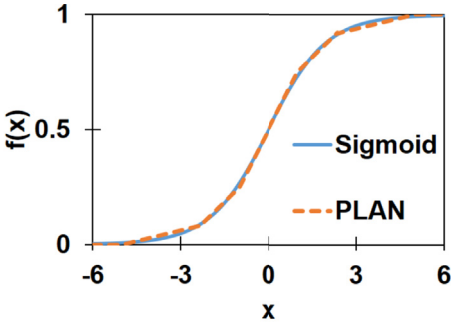
### 4.4. Evaluation for higher dimensional datasets

We further evaluate the effectiveness of LAM with four other higher-dimensional datasets. Also, to gain more insight into training methodology, we compare approximation strategies that individually adopt accurate (EFM) and/or approximate (LAM) multipliers in training and testing phases, respectively. Fig. 14 illustrates the four strategies we evaluated.

Fig. 15 shows the training results for the four datasets. To test the compatibility of LAM to BWS, we varied the number of fraction bits as 10, 16, and 23 while the bit width for exponent remains 8 bits to keep the dynamic range. The results show that either case attains a similar accuracy both for training and testing sets. When looking into the details, ISOLET with 23 bit and CNAE-9 with 16 bit in testing set lose 1% accuracy, probably because training with LAM was trapped into overfitting. On the other hand, in different combinations of dataset and fraction bit width, Cases #2 to #4 with LAM provide very close or even better classification accuracy compared to Case #1 only with EFM. Another observation is that the differences between Case #2 to Case #4 are also small. Overall, these results reveal that there is no reason to fully or partially use EFM in training, at least for the databases used in our

| **PLAN** $f(x), f(-x) = 1 - f(x)$ | |
|---|---|
| **Criterion** | **Formula** |
| $x \geq 5$ | 1 |
| $2.375 \leq x < 5$ | $x/32 + 0.84375$ |
| $1 \leq x < 2.375$ | $x/8 + 0.625$ |
| $0 \leq x < 1$ | $x/4 + 0.5$ |

(a)




(b)

**Fig. 10.** PLAN function, an approximate form of sigmoid function [29]. (a) Expression of PLAN and (b) Plot of original sigmoid and PLAN functions.

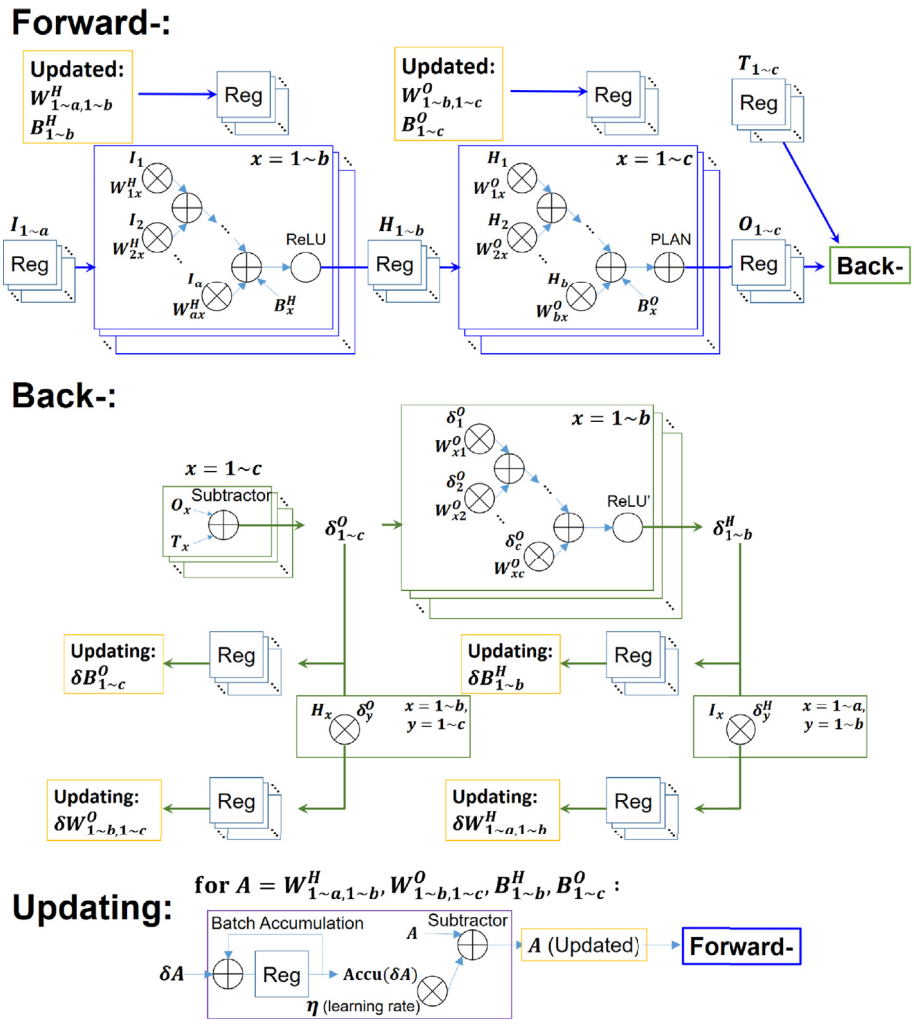



**Fig. 11.** Diagram of our dedicated NN training hardware, including forward-, back-, and updating blocks. ⊗ and ⊕ denote multipliers and adders, respectively. All the ⊗ and ⊕ are spatially implemented at the same time.

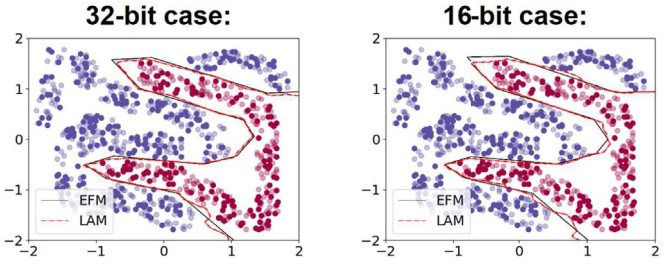



**Fig. 12.** Boundaries trained for 2-D classification FOURCLASS.

experiment. LAM can provide trained NN models whose classification accuracy is comparable to that of those trained by EFM.

We next evaluate the hardware cost. For estimating the hardware improvement thanks to LAM, a simple projection is performed based on the FOURCLASS results. The power consumption of the training engine is estimated by the numbers of arithmetic units and registers. The power values of each arithmetic unit and register are obtained from the logic synthesis result. The remaining thing is to count the number of arithmetic units and registers referring to Fig. 11. Table 2 lists our statistics for the four datasets, indicating that the usage of three parts closed to 1:1:1.

As the number of multipliers, adders, and registers increase in higher-dimensional datasets, the power improvement rate by LAM for training a larger NN-size structure is roughly equal or slightly larger than that of FOURCLASS. Meanwhile, as a conservative estimate, the least improvement rate is accessed based on the FOURCLASS benefit. Table 3 shows the projected values. Here, we align their synthesized frequency, and in this particular estimation the power analysis is not annotated with actual switching activity (i.e. vectorless power). Terminology "EFM" is EFM-32bit, "LAM" represents LAM-32bit, "BWS" denotes EFM-19bit, and then "LAM + BWS" means LAM-19bit. Here, EFM-19bit and LAM-19bit adopt 8 bits for the exponent and 10 bits for the fraction, and they have the same fraction bit width as conventional 16-bit format. From Table 3, LAM attains 2.5X and LAM + BWS achieves 4.9X power efficiency, where 2.2X originates from LAM.

In this section, the power efficiency is evaluated based on estimation. In the next section, we are going to demonstrate the power efficiency improvement through hardware measurement.

## 5. Evaluation in GPU design

Following the performance evaluation with dedicated training engines in the previous section, this section applies LAM and BWS to an open-source GPU design and clarifies the advantage in NN training.

### 5.1. LAM-based GPU implementation on FPGA

To train large NNs, training engines demand programmability since

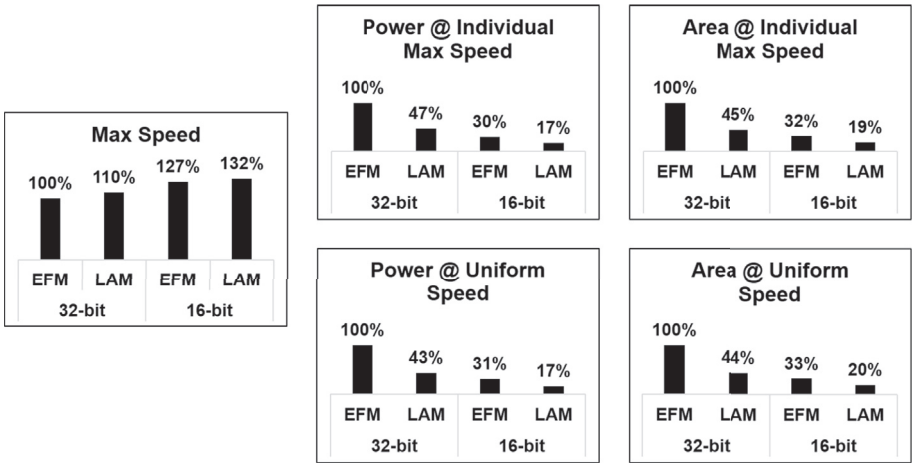
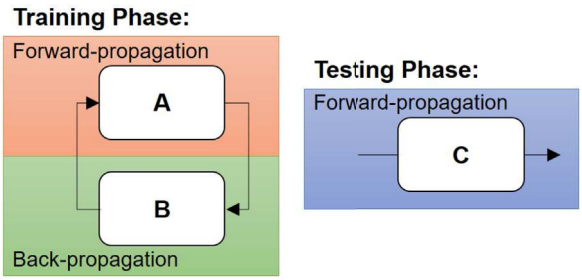



**Fig. 13.** Speed, power, and area comparisons between synthesized LAM-based and EFM-based training engines. The synthesis setup is identical to that of Fig. 9, and all the values are normalized by those of EFM 32-bit case.




| Multiplier Type | Case #1 | Case #2 | Case #3 | Case #4 |
|---|---|---|---|---|
| A | EFM | LAM | EFM | LAM |
| B | | | | EFM |
| C | | | LAM | LAM |

**Fig. 14.** Different approximation strategies that use EFM and/or LAM in training and testing phases.

the entire datapath cannot be implemented spatially in a chip at once and temporal sharing for, e.g., each layer and each kernel becomes indispensable. Then, we select Nyuzi, which is an open-source processor for GPGPU applications [30], as the baseline design and incorporate LAM and BWS with Nyuzi. Nyuzi is distributed as a synthesizable RTL Verilog with an instruction set emulator, and a C/C++ compiler.

Fig. 16 shows the power evaluation setup. To proceed with our experiment, we modified the RTL code to integrate LAM and BWS functionality. For performing power measurement on hardware, we synthesized the original and modified RTL codes by Intel Quartus with 50 MHz clock frequency targeting Terasic DE2-115 evaluation board. Table 4 lists the logic element counts for each case after Quartus synthesis. Here, the bit-widths of the cases "EFM", "LAM", "BWS", and "LAM + BWS" are aligned with those adopted in Table 3. From the table, the hardware design embedded with LAM and LAM + BWS saves about 12K and 14.3K logic elements compared to EFM, respectively, which are mainly contributed from FPU blocks.

We also prepared a C-program code for NN training. The implemented code is compiled and the binary code is loaded in Nyuzi on FPGA. Then, we launch the NN training program on Nyuzi and measure the power consumption of FPGA. Fig. 17 shows the hardware setup for measuring power dissipation. Agilent N6705A DC power analyzer is used to provide 12 V power supply of DE2-115 board, and it also serves as a power meter. In this setup, the measured power includes not only the

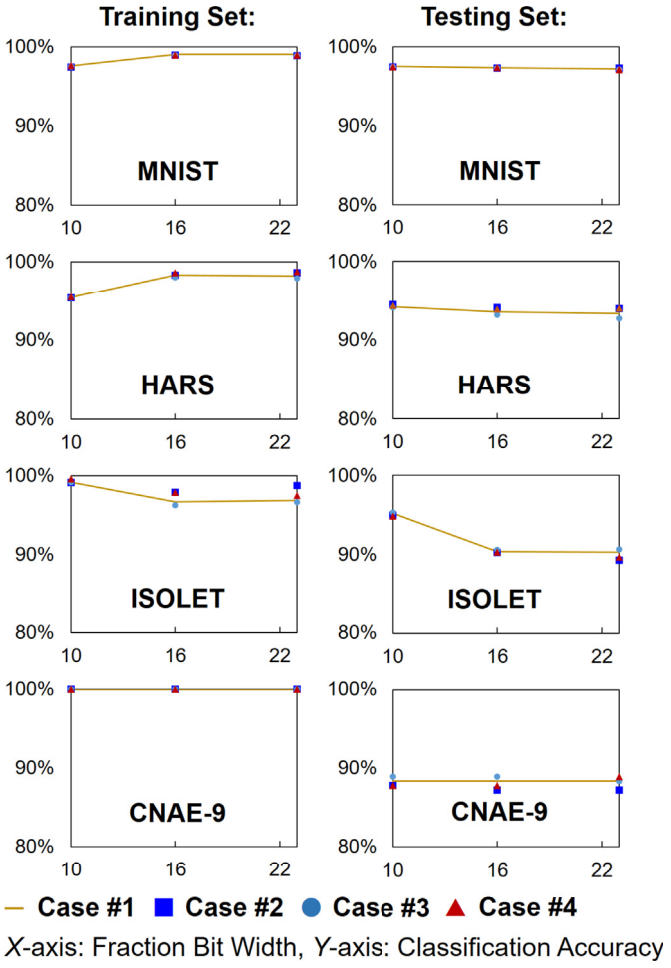



**Fig. 15.** Training results for MNIST, HARS, ISOLET, CNAE-9 under different approximation strategies illustrated in Fig. 14 for the fraction bit widths of 10, 16, and 23.

power of Nyuzi on FPGA but also that of other peripheral circuitry on the board.

### 5.2. Measurement results

Fig. 18 shows the measured transient power consumption during

**Table 2**
Statistics for usage of multipliers, adders, and registers in our dedicated NN hardware.

| Dataset | Usage | | |
|---|---|---|---|
| | #Multipliers | #Adders | #Registers |
| FOURCLASS | 122 | 128 | 142 |
| MNIST | 372,310 | 372,340 | 371,050 |
| HARS | 68,326 | 68,344 | 69,352 |
| ISOLET | 195,626 | 195,704 | 194,664 |
| CNAE-9 | 260,509 | 260,536 | 261,657 |

**Table 3**
Power estimation results for training larger-size of NN (projected from FOURCLASS benchmarking result).

| Power (Normalized to EFM) | | | |
|---|---|---|---|
| EFM | LAM | BWS | LAM + BWS |
| 100% | 40% | 37% | 20% |

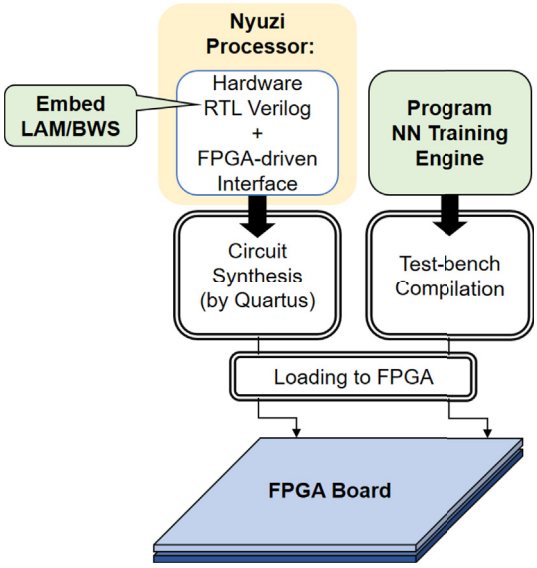



**Fig. 16.** Flow for measuring power of LAM (and LAM + BWS) based Nyuzi on FPGA.

**Table 4**
Number of logic elements used to implement Nyuzi on FPGA.

| | # of used logic elements | | | |
|---|---|---|---|---|
| | EFM | LAM | BWS | LAM + BWS |
| Overall | 82,989 | 71,023 | 71,909 | 68,681 |
| FPU | 33,056 | 21,224 | 21,545 | 18,342 |

Nyuzi operation. The waveform is divided into three stages according to the Nyuzi operation status, and the red line in Fig. 18 represents the average value in each stage. Referring to the assembly code, the period (3) executes only the "NOP" operation. Thus, we suppose that the difference in average power between period (2) and period (3) represents the power necessary for NN training.

Fig. 19 firstly shows the measured power of NN training for FOURCLASS dataset. Again, the definition of EFM, LAM, BWS, LAM + BWS are all consistent with Table 3. Here, the figure includes the results for

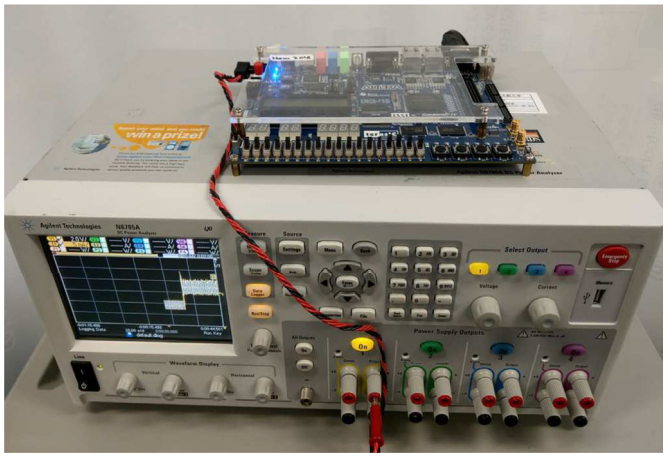


**Fig. 17.** Photo for power measurement setup of Nyuzi processor. The FPGA board is placed on the DC power analyzer. The power analyzer is used for voltage supply and power measurement.

different programming styles; single-thread and multi-thread. The power values for each style are normalized by that of EFM case with the same style. The results show that LAM and BWS are effective in power saving irrelevant to the programing styles.

Fig. 20 shows the measurement results for four higher-dimensional datasets. Again, we obtained a similar amount of power reduction for all the datasets. LAM-based floating-point training computation achieves 24%–28% power improvement, and the improvement increases to 35%–41% in the LAM + BWS case, which corresponds to 1.32X or higher power efficiency.

## 6. Evaluation for deeper NNs

The advantage of applying LAM and BWS to deeper NN training is presented in this section. We show the training results of NNs with 2, 3, and 4 hidden layers, respectively, for MNIST dataset. In every NN structure, each hidden layer consists of 50 neurons. The NN training results for adopting solo-EFM and solo-LAM with different configurations of BWS are shown in Fig. 21.

Fig. 21 shows that, up to 4-hidden-layer NN, LAM-based training yields the accuracy comparable to that of EFM-based training as long as

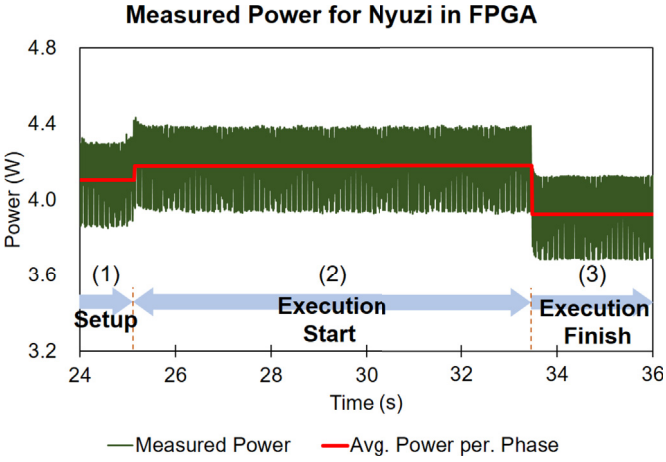



**Fig. 18.** Transient power response measured during Nyuzi operation. Phase (2) is the phase for executing NN training program. In Phase (3), the training process already finished.

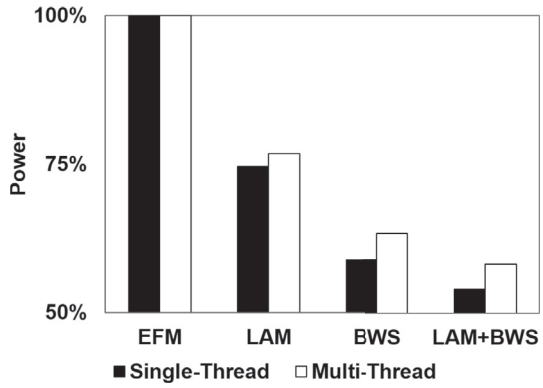



**Fig. 19.** Power measurement results for single-thread and multi-thread (FOURCLASS dataset). All the values are normalized to EFM case.

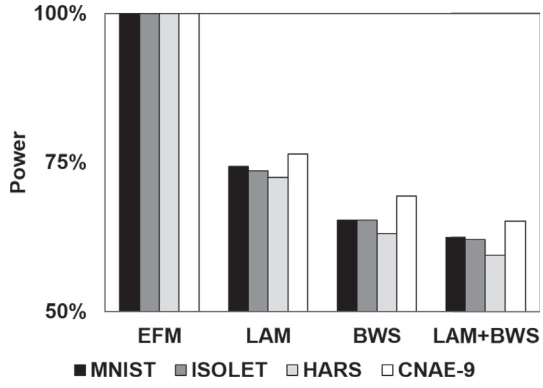



**Fig. 20.** Power measurement results for MNIST, HARS, ISOLET, CNAE-9 datasets. All the values are normalized to EFM case.

the result of EFM-based training with BWS is reasonably accurate. When 97% accuracy is considered acceptable (above 10 fraction bits), the accuracy drop contributed by LAM is less than 0.3%. This result indicates that the training accuracy is primarily determined by BWS instead of LAM, and LAM is applicable even when aggressive BWS is implemented. Note that the absolute accuracy could be improved with more sophisticated NNs, such as CNN [11,35]. The results for other datasets of HARS, ISOLET, and CNAE-9 show a similar trend as MNIST, and hence the detailed results are omitted here. In overall, the average and maximum accuracy drops by LAM for the cases other than MNIST are mere 0.1% and 2.2% when 94% accuracy is considered acceptable, and LAM outperforms at 31% points in all the cases we run.

Fig. 22 shows the training curve for MNIST dataset. For the sake of clarity, we plot only one scenario for the NN with 4-hidden layers and 23 fraction bits. As shown in the figure, at each epoch from 1 to 40, there is no significant difference in classification accuracy between LAM-based and EFM-based training. This result indicates that adopting LAM in NN training does not require additional processing time to reach the same accuracy, and thus, at least within 4-hidden-layer NNs, training completely relying on LAM is qualified and EFM is not necessary.

## 7. Conclusions

Edge computing drives the demands for more power-efficient data processing, such as neuron network training. This work evaluated whether approximate floating-point multiplier, which can cover a broad dynamic range, could be adopted in NN training achieving higher energy efficiency. Specifically, we focused on logarithm-approximate multiplier

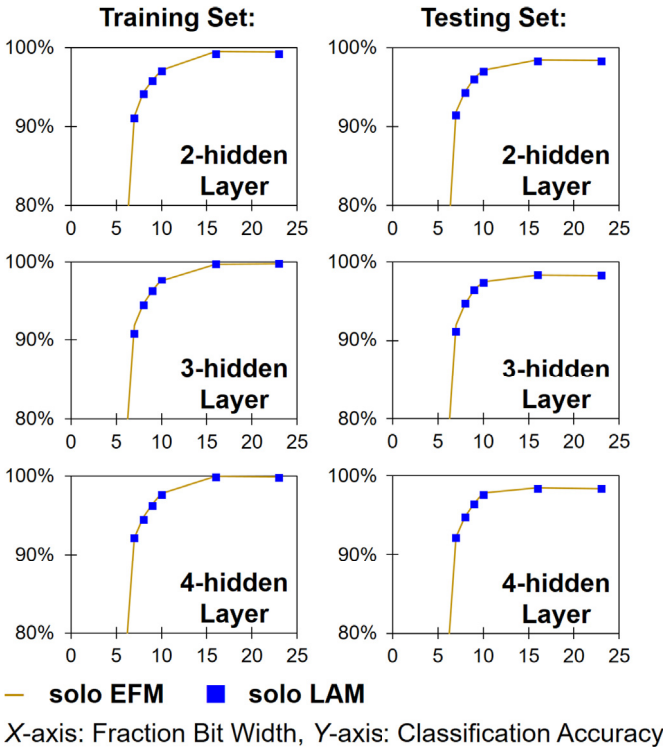



**Fig. 21.** MNIST training results for NNs having 2, 3, and 4 hidden layers and various fraction bits. The attained accuracies are almost identical.

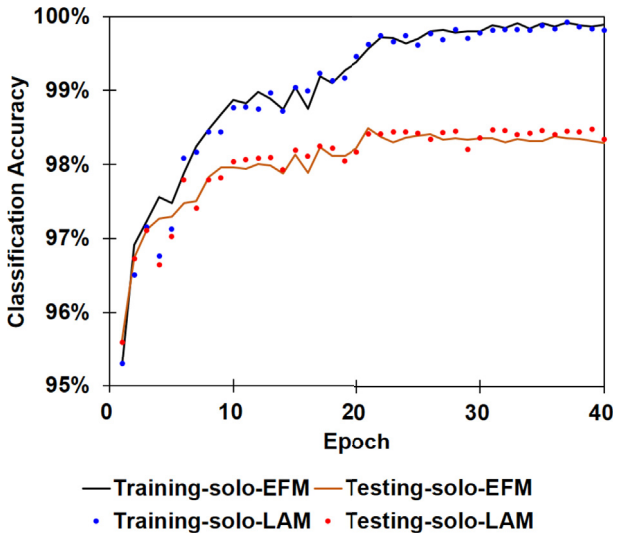



**Fig. 22.** Training curves of 4-hidden-layer NNs with 23 fraction bits. The speeds of EFM-based and LAM-based training are almost identical.

(LAM) incorporating bit-width scaling (BWS) to reduce primary MAC computation complexity. The experimental results with dedicated hardware design show that training NNs with LAM can achieve 10% speed-up and 2.3X power reduction in addition to 2.3X area saving as well at the same speed when training a 2-D classification dataset. Even when training with LAM + BWS, there is no more than 1.0% accuracy discrepancy compared with the exact multiplier, where LAM + BWS outperforms, rather than degrades, the accuracy more frequently. As for the hardware performance, 4.9X energy efficiency is attained, where

2.2X originates from LAM. We further quantified LAM effectiveness with an open-source GPU design. The power reduction was evaluated with the FPGA hardware measurement. We confirmed 1.32X power efficiency improvement in the LAM-based GPU design compared with the EFM-based GPU design. Finally, LAM and LAM + BWS are experimentally qualified to be applicable to training up to 4 hidden layers, even with aggressive BWS.

## Declaration of competing interest

None.

## CRediT authorship contribution statement

**TaiYu Cheng:** Conceptualization, Software, Validation, Formal analysis, Data curation, Writing - original draft. **Yukata Masuda:** Resources. **Jun Chen:** Methodology. **Jaehoon Yu:** Supervision. **Masanori Hashimoto:** Supervision, Writing - review & editing.

## References


[1] J. Chen, X. Ran, Deep learning with edge computing: a review, Proc. IEEE 107 (8) (2019) 1655–1674.
[2] G.L. Pedro, et al., Edge-centric computing: vision and challenges, in: ACM SIGCOMM Computer Communication Review, vol. 45, Oct. 2015, pp. 37–42, no. 5.
[3] P. Grulich, et al., Collaborative edge and cloud neural networks for real-time video processing, in: Proc. VLDB Endowment, vol. 11, 2018, pp. 2046–2049.
[4] Y. Huang, et al., When deep learning meets edge computing, in: ICNP, Dec. 2012, pp. 1–2.
[5] A. Krizhevsky, et al., Imagenet classification with deep convolutional neural networks, in: NIPS, Dec. 2012, pp. 1097–1105.
[6] J.Y.F. Tong, et al., Reducing power by optimizing the necessary precision/range of floating-point arithmetic, TVLSI 8 (3) (2000) 273–286.
[7] S. Venkataramani, et al., Axnn: energy-efficient neuromorphic systems using approximate computing, in: ISLPED, Aug. 2014, pp. 27–32.
[8] Q. Zhang, et al., Approxann: an approximate computing framework for artificial neural network, in: DATE, Mar. 2015, pp. 701–706.
[9] J. Kung, et al., A power-aware digital feedforward neural network platform with backpropagation driven approximate synapses, in: ISLPED, Jul. 2015, pp. 85–90.
[10] J. David, et al., Training Deep Neural Networks with Low Precision Multiplications, 2014 arXiv:1412.7024.
[11] N. Wang, et al., Training deep neural networks with 8-bit floating point numbers, in: Proc. NIPS, Dec. 2018, pp. 7685–7694.
[12] D. Kim, et al., A power-aware digital multilayer perceptron accelerator with on-chip training based on approximate computing, TETIC 5 (2) (Feb. 2017) 164–178.
[13] Simons et al., A review of binarized neural networks, Electronics 8. 661. 10.3390.
[14] M. Horowitz. Energy Table for 45nm Process. Stanford VLSI wiki.
[15] T. Cheng, et al., Minimizing power for neural network training with logarithm-approximate floating-point multiplier, in: PATMOS, Jul. 2019.
[16] M. Gao, et al., Energy efficient runtime approximate computing on data flow graphs, in: ICCAD, Nov. 2017, pp. 444–449.
[17] C. Chang, C. Lin, Fourclass, 1996. https://www.csie.ntu.edu.tw/cjlin/libsvmtools/datasets/binary.html.
[18] S. Haykin, Neural Networks and Learning Machines, 3 edition, PEARSON Education, 2008.
[19] K. Ueyoshi, et al., QUEST: a 7.49TOPS multi-purpose log-quantized DNN inference engine stacked on 9MB 3D SRAM using inductive coupling technology in 40nm CMOS, in: ISSCC, Feb. 2018, pp. 186–196.
[20] U. Lotrič, et al., Applicability of approximate multipliers in hardware neural networks, Neurocomputing 96 (Nov. 2012) 57–65.
[21] M. Imani, et al., RMAC: runtime configurable floating point multiplier for approximate computing, in: ISLPED, Jul. 2018.
[22] M. Imani, et al., CANNA: neural network acceleration using configurable approximation on GPGPU, in: ASPLOS, 2018.
[23] J. Johnson, Rethinking Floating Point for Deep Learning, 2018. arXiv: 1811.01721v1.
[24] Y. LeCun, et al., Mnist, 1998. http://yann.lecun.com/exdb/mnist.
[25] Uci Machine Learning Repository, Human Activity Recognition Using Smartphones Data Set, 2012. http://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones.
[26] Uci Machine Learning Repository, ISOLET Data Set, 1994. http://archive.ics.uci.edu/ml/datasets/ISOLET.
[27] Uci Machine Learning Repository, CNAE-9 Data Set, 2009. http://archive.ics.uci.edu/ml/datasets/CNAE-9.
[28] K. Murphy, Machine Learning, MIT Press, Cambridge, Mass, 2012.
[29] A. Tisan, et al., Digital Implementation of the Sigmoid Function for FPGA Circuits, Acta Technica Napocensis Electronics and Telecommunications, 2009.
[30] Jeff Bush, NyuziProcessor: Source Code, 2015, in: https://github.com/jbush001/NyuziProcessor.
[31] E.H. Lee, et al., LogNet: energy-efficient neural networks using logarithmic computation, in: ICASSP, 2017, pp. 5900–5904.
[32] D. Miyashita, et al., Convolutional Neural Networks Using Logarithmic Data Representation, 2016 arXiv:1603.01025.
[33] M.S. Kim, et al., "Efficient Mitchell's approximate log multipliers for convolutional neural networks, TOC 68 (5) (2019) 660–675.
[34] W. Liu, et al., Design and evaluation of approximate logarithmic multipliers for low power error-tolerant applications, TCAS-I 65 (9) (2018) 2856–2868.
[35] R. DiCecco, et al., FPGA-based training of convolutional neural networks with a reduced precision floating-point library, in: ICFPT, 2017, pp. 239–242.


**TaiYu Cheng** received the B.E. and M.E. degrees in electrical engineering from National Taiwan University, Taipei, Taiwan, in 2010 and 2012, respectively. From 2012 to 2018, he was with Taiwan Semiconductor Manufacturing Company, Hsinchu, Taiwan, where he has been engaged in design flow of timing closure. Since 2018, he has been a Ph. D. student with the Department of Information Systems Engineering, Osaka University, Osaka, Japan. His research interests include low-power circuit design.

**Yutaka Masuda** received the B.E., M.E., and Ph.D. degrees in Information Systems Engineering from the Osaka University, Osaka, Japan, in 2014, 2016, and 2019, respectively. He is currently an Assistant Professor in Center for Embedded Computing Systems, Graduate School of Informatics, Nagoya University. His research interests include low-power circuit design. He is a member of IEEE, IEICE, and IPSJ.

**Jun Chen**received the B.E. and M.E. degrees in control theory and engineering from Tongji University, Shanghai, China, in 2004 and 2007, respectively, and received his Ph.D. degree in Information Systems Engineering from the Osaka University, Osaka, Japan, in 2020. From 2008 to 2016, he was with Synopsys Inc., Shanghai, China, where he has been engaged in research and development of routing congestion, power and placement optimization flow. He is currently a software engineer at Giga Design Automation Co., Ltd. His research interests include computer-aided-design for digital integrated circuits, power and signal integrity analysis..

**Jaehoon Yu** received his B.E. degree in Electrical and Electronic Engineering and his M.S. degree in Communications and Computer Engineering from Kyoto University, Kyoto, Japan, in 2005 and 2007, respectively, and received his Ph.D. degree in Information Systems Engineering from Osaka University, Osaka, Japan, in 2013. From 2013 to 2019, he was an assistant professor at Osaka University. He is currently an associate professor at Tokyo Institute of Technology, Japan. His research interests include computer vision, machine learning, and system-level design. He is a member of IEEE, IEICE, and IPSJ.

**Masanori Hashimoto** received the B.E., M.E. and Ph.D. degrees in communications and computer engineering from Kyoto University, Kyoto, Japan, in 1997, 1999, and 2001, respectively. He is currently a Professor with the Department of Information Systems Engineering, Graduate School of Information Science and Technology, Osaka University, Suita, Japan. His current research interests include the design for manufacturability and reliability, timing and power integrity analysis, reconfigurable computing, soft error characterization, and low-power circuit design. Dr. Hashimoto was a recipient of the Best Paper Awards from ASP-DAC in 2004 and RADECS in 2017, and the Best Paper Award of the IEICE Transactions in 2016. He was on the Technical Program Committee of international conferences, including DAC, ICCAD, ITC, Symposium on VLSI Circuits, ASP-DAC, and DATE. He serves/served as an Associate Editor for the IEEE Transactions on VLSI Systems, IEEE Transactions on Circuits and Systems I, ACM Transactions on Design Automation of Electronic Systems, and Elsevier Microelectronics Reliability.